AUTSEG: Automatic Test Set Generator for Embedded Reactive Systems

Mariem Abdelmoula, Daniel Gaffé, Michel Auguin

AUTSEG: Automatic Test Set Generator for Embedded Reactive Systems

Mariem Abdelmoula, Daniel Gaffe, and Michel Auguin

LEAT, University of Nice-Sophia Antipolis, CNRS
930 route des Colles, BP 145, 06903 Sophia Antipolis Cedex France
Mariem.Abdelmoula@unice.fr
Daniel.Gaffe@unice.fr
Michel.Auguin@unice.fr

Abstract. One of the biggest challenges in hardware and software design is to ensure that a system is error-free. Small errors in reactive embedded systems can have disastrous and costly consequences for a project. Preventing such errors by identifying the most probable cases of erratic system behavior is quite challenging. In this paper, we introduce an automatic test set generator called AUTSEG. Its input is a generic model of the target system, generated using the synchronous approach. Our tool finds the optimal preconditions for restricting the state space of the model. It only works locally on significant subspaces. Our approach exhibits a simpler and efficient quasi-flattening algorithm than existing techniques and a useful compiled form to check security properties and reduce the combinatorial explosion problem of state space. To illustrate our approach, AUTSEG was applied to the case of a transportation contactless card.

Keywords: AUTSEG, Test Sets, State Machines, States Space Covering, Sequences Generation, Contactless Smart Card, Specification, Synchronous Model.

1 Introduction

Verifying automatically and formally that a system is working correctly is not trivial nowadays due to the increasing complexity of computer programs and their strong interaction with the environment. An important class of systems facing such problems are reactive systems. They continuously react and respond to their environment. Reactive systems belong to the large family of FSMs (Finite State Machines). They are ubiquitous in everyday life, varying from simple thermostats to the control of nuclear power plants, avionics, telesurgery, and online payment. Security for these systems is critical; even minor errors are unacceptable. In this paper, we focus on verification of embedded software controlling the reactive system behavior. To illustrate our approach, we aim to verify the implementation of the OS integrated in a contactless smart card for transportation. We specifically target the verification of the card's functionality and

security features. Smart Cards are ubiquitous, with more than 200 million used across the globe for transportation, telephony, health insurance, banking, ID, etc. Frauds are especially critical for banking cards, as counterfeiters are able to exploit the vulnerabilities of coding systems.

Furthermore, the card's complexity makes it difficult for a human to identify all possible sensitive situations or to validate it by classical methods. We need approximately 500 000 years to test the first 8 bytes if we consider a classical Intel processor that generates 1000 test sets per second. As well, combinatorial explosion of possible modes of operation makes it nearly impossible to attempt a comprehensive simulation. The problem is exacerbated when the system integrates data processing, so results have significant effects on system behavior. Thus, we strive for highly-automated testing techniques. We aim as well to describe with symbolic means the reachable state space model of the card specification for a particular security property. Expertise in symbolic verification and synchronous languages are required to automatically generate exhaustive test sets that represent critical situations.

We hence focus on a set of techniques known as formal methods, based on the Binary Decision Diagram BDD [1] that are used in computer science to ensure correct system behavior. We propose in this paper, an automatic test set generator called AUTSEG. Generating automatic test sets and covering all transitions is not a new research area. However, we notably address in this paper the state space explosion problem. We first generate a powerful quasi-flattening algorithm which performs a simple and deterministic model, thus facilitating code generation. We qualify by a second algorithm the correct behavior of the global system, without requiring coverage of all system states and transitions.

In the remainder of this paper, we give an overview of related work in Section 2. We present in Section 3 our global approach for test generation. AUTSEG and details on its capabilities are presented in Section 4. Section 5 presents the application of our generator to a specific contactless card for transportation. Experimental results are shown in Section 6. Finally, Section 7 concludes the paper with some directions for future work.

2 Related Work

Lutess V2 [2] is a test environment, written in Lustre, for synchronous reactive systems. It automatically generates tests that dynamically feed the program under test from the formal description of the program environment and properties. This version of Lutess deals with numeric inputs and outputs unlike the first version [3]. Lutess V2 is based on Constraint Logic Programming (CLP) and allows the introduction of hypotheses to the program under test. Due to CLP solvers' capabilities, it is possible to associate occurrence probabilities to any Boolean expression. However, this tool requires the conversion of tested models to the Lustre format, which may cause a few issues in our tests.

B.Blanc presents in [4] a structural testing tool called GATeL, also based on CLP. GATeL aims to find a sequence that satisfies both the invariant and the

test purpose by solving the constraints problem on program variables. Contrary to Lutess, GATeL interprets the Lustre code and starts from the final state and ends with the first one. This technique relies on human intervention, which is stringently averted in our paper.

C.Jard and T.Jeron, present TGV (Test Generation with Verification technology) in [5], a powerful tool for test generation from various specifications of reactive systems. It takes as inputs a specification and a test purpose in IOLTS (Input Output Labeled Transition System) format and generates test cases in IOLTS format as well. TGV allows three basic types of operations: 1. It identifies sequences of the specification accepted by a test purpose, based on the synchronous product; 2. It then computes visible actions from abstraction and determinization; 3. Finally, it selects test cases by computation of reachable states from initial states and co-reachable states from accepting states. A limitation lies in the non-symbolic (enumerative) dealing with data. The resulting test cases can be big and therefore relatively difficult to understand.

D.Clarke extends this work in [6], presenting a symbolic test generation tool called STG. It adds the symbolic treatment of data by using OMEGA tool capabilities. Test cases are therefore smaller and more readable than those done with enumerative approaches in TGV. STG produces the test cases from an IOSTS specification (Input Output Symbolic Transition System) and a test purpose. Despite its effectiveness, this tool is no longer maintained.

[7] describes STS (Symbolic Transition Systems), quite often used in systems testing. STS enhances readability and abstraction of behavioral descriptions compared to formalisms with limited data types. STS also addresses the state explosion problem through the use of guards and typed parameters related to the transitions. At the moment, STS hierarchy does not appear very enlightening outside the world of timed/hybrid systems or well-structured systems. Such systems are outside of the scope of this paper.

ISTA (Integration and System Test Automation) [8] is an interesting tool for automated test code generation from High-Level Petri Nets. ISTA generates executable test code from MID (Model Implementation Description) specifications. Petri net elements are then mapped to implementation constructs. ISTA can be efficient for security testing when Petri nets generate threat sequences. However, it focuses solely on liveness properties checking, while we focus on security properties checking.

J.Burnim presents in [9], a testing tool for C called CREST. It inserts instrumentation code using CIL (C Intermediate Language) into a target program. Symbolic execution is therefore performed concurrently with the concrete execution. Path constraints are then solved using the YICES solver. CREST currently reasons symbolically only about linear, integer arithmetic. Closely related to CREST, KLOVER [10] is a symbolic execution and automatic test generation tool for C++ programs. It basically presents an efficient and usable tool to handle industrial applications. Both KLOVER and CREST cannot be adopted in our approach, as they accommodate tests on real systems, whereas we target tests on systems still being designed.

3 Global Process

Let us start with a description of the global architecture we have designed for our test. Fig.1 shows 4 main operations explained in detail in the following sections.

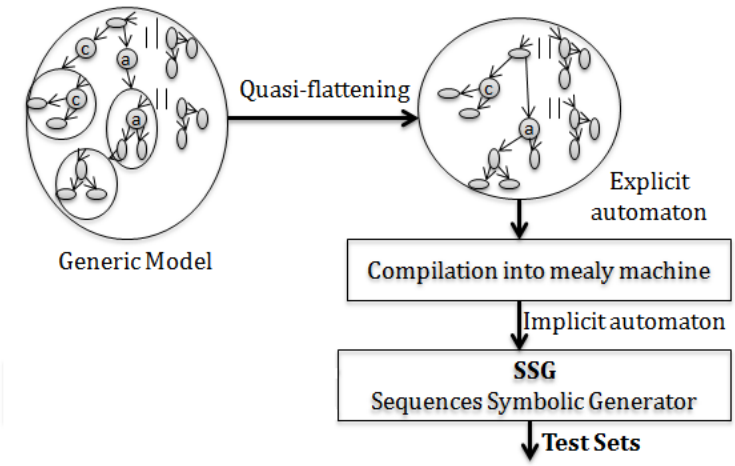

Fig. 1. Global process structure

1. Generic model: it presents the principal input of our test. The global architecture is composed of hierarchical and parallel concurrent FSM based on the synchronous approach. It should conform to the specification of the system under test.

2. Quasi-flattening process: we flatten only hierarchical automata, the rest of automata remaining parallel. This offers a simple model and brings more flexibility to identify all possible evolutions of the system.

3. Compilation process: it generates an implicit automaton represented by a Mealy machine from an explicit automaton. This process compiles the model, checks the determinism of all automata and ensures the persistence of the system behavior.

4. SSG (Sequences Symbolic Generator): it extracts necessary preconditions which lead to specific, significant states of the system from generated sequences.

3.1 Quasi-flattening process

The straightforward way to analyze a hierarchical machine is to flatten it (recursively substitute in a hierarchical FSM each super state with its associated FSM), then apply as an example a model-checking tool on the resulting FSM. Let's consider the model shown in Fig.2, which shows automata interacting and communicating between each other. Most of them are sequential, hierarchical

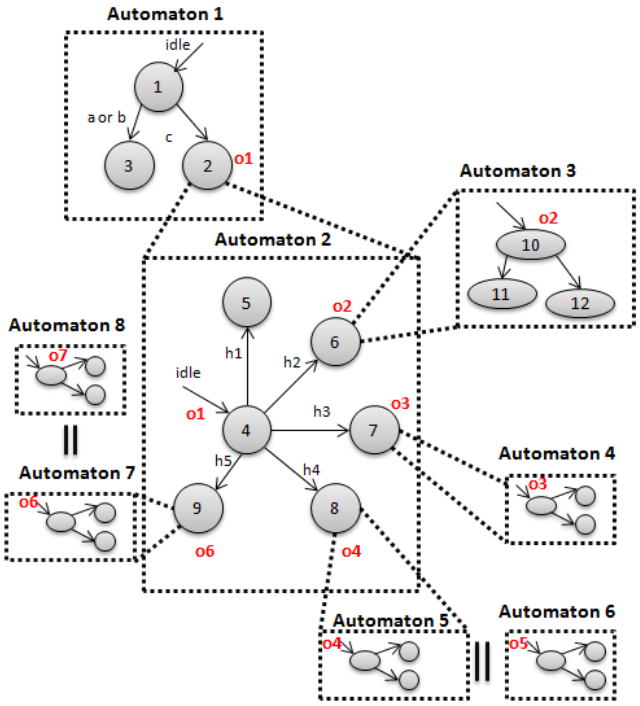

Fig. 2. Model Design

automata (e.g. automata 1 and 2), while others are parallel automata (e.g. automata 6 and 8). We note in this architecture 13122 ($3 \times 6 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$) possible states derived from parallel executions (graphs product) while there are many fewer active states at once. Indeed, this model is designed by the graphical formalism SyncChart [11]. A classical analysis is to transform this hierarchical structure to the synchronous language Esterel [12]. Such transformation is not quite optimized. Furthermore, Esterel is not able to realize that there is only one active state at once. In practice, compiling such structure by Esterel generates 83 registers making roughly 9.6×10^{24} states. Hence, the behoof of our process. Opting for a quasi-flattening, we have flattened only hierarchical automata. Thus, state 2 of automaton 1 is substituted by the set of states 4,5,6,7,8,9 of automaton 2 and so on. Required transitions are rewritten thereafter. Parallel automata are acting as observers that manage the model's control flags. Flattening parallel FSMs explodes usually in number of states. Thus there is no need to flatten them, as we can compile them separately, then concatenate them with the flat model retrieved at the end of the compilation process.

Algorithm 1 details our quasi-flattening operation. We denote downstream the initial state of a transition and upstream the final one. This algorithm implements three main operations. Overall, It replaces each macro state with a corresponding FSM. It first interconnects the internal initial states. It then re-

Algorithm 1 Flattening operation

$St \leftarrow$ State; $SL \leftarrow$ State List of FSM; $t \leftarrow$ transition in FSM
while $SL \neq$ empty **do**
  Consider each St from SL
  if (St is associated to a sub-FSM) **then**
    mark the deletion of St
    load all *sub-St* from sub-FSM (particularly init-*sub-St*)
    for (all t of FSM) **do**
      if (upstream(t) == St) **then**
        upstream(t) $\leftarrow$ init-*sub-St* // illustration in Fig.3 ($t0$, $t1$, $t2$ relinking)
    for (all t of FSM) **do**
      if (downstream(t) == St) **then**
        if (t is a normal-term transition) **then**
          // illustration in Fig.5
          for (all *sub-St* of sub-FSM) **do**
            if (*sub-St* is associated to a sub-sub-FSM) **then**
              create t' (*sub-St*, upstream(t)) // Keep recursion
            if (*sub-St* is final) **then**
              for (all t'' of sub-FSM) **do**
                if (upstream(t'') == *sub-St*) **then**
                  upstream(t'') $\leftarrow$ upstream(t)
                  merge effect(t) to effect(t'')
              mark the deletion of *sub-St*
        else
          // normal transition: illustration in Fig.3
          // For example $t3$ is less prior than $t6$ and replaced by $\overline{t6}.t3$ and $t6$
          for (all *sub-St* of sub-FSM) **do**
            create t'(*sub-St*,upstream(t),trigger(t),effect(t))
            for (all *sub-t* of sub-FSM) **do**
              turn-down the *sub-t* priority (or turn up t' priority)
          delete t
    add and rename all *sub-t* transitions from subFSM to SL
    add and rename all *sub-St* state from subFSM to SL
    cancel marked states

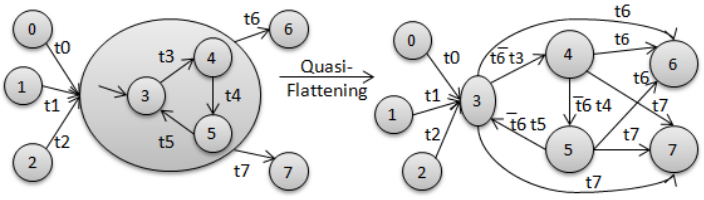

Fig. 3. Initial and Normal Transition Linking

places normal [1] terminations with internal transitions in a recursive manner. Finally, it interconnects all states of the internal FSM.

[1] Refers to SyncCharts "normal termination" transition [11]

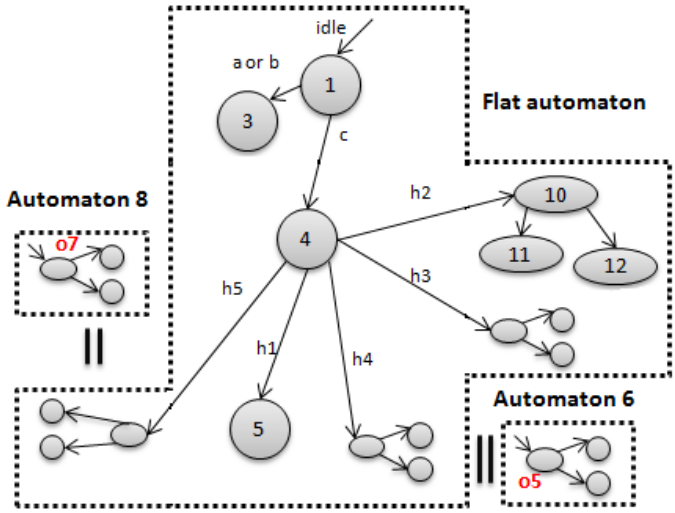

Fig. 4. Flat Model

Flattening the hierarchical model of Fig.2 results in a flat structure shown in Fig.3.1. As the activation of state 2 is a trigger for state 4, these two states will be merged, just as state 6 will be merged to state 10, etc. Automata 6 and 8 (observers) remain parallel in the expanded automaton; they are small and do not increase the computational complexity. The model in Fig.3.1 contains now only 144 ($16 \times 3 \times 3$) state combinations. In practice, compiling this model according to our process generates merely 8 registers, equivalent to 256 states.

Our flattening differs substantially from those of [13] and [14]. We assume that a transition, unlike the case of statecharts, cannot exit different hierarchical levels. Several operations are thus executed locally, not on the global system. This yields a simpler algorithm and faster compilation. To this end, we have integrated the following assumptions in our algorithm:
-Normal termination. Fig.5 shows an example of normal termination carried when a final internal state is reached. It allows a unique possible interpretation and facilitates code generation.
-Strong preemption. Unlike classical preemption, internal outputs of the preempted state are lost during the transition.

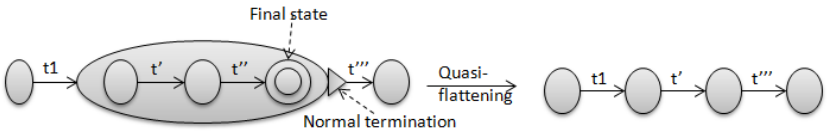

Fig. 5. Terminal Transition Linking

3.2 Compilation process

We proceed in our approach to a symbolic compilation of the model into a Mealy machine, implicitly represented by a set of Boolean equations (circuit of logic gates and registers presenting the state of the system). Compiling an explicit automaton into an implicit one is a well-known process in hardware design. Classical works use the *one-hot* representation [15], while our compilation requires only $log_2(nbstates)$ registers. Actually, concurrent automata and flat automata are compiled separately. Compilation results of these automata are concatenated at the end of this process. They are represented by an union of sorted equations rather than a Cartesian product of graphs to support the synchronous parallel operation and instantaneous signals diffusion. Accordingly, we note a substantial reduction on the size of tested system.

3.3 SSG (Sequences Symbolic Generator)

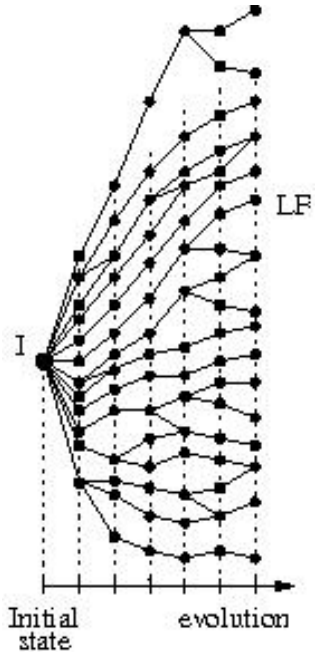

Fig. 6. Classical Sequences Generation

We explain in this section the process we follow to automatically generate symbolic sequences of test sets. As mentioned before, we seek to restrict the states space and confine only to significant states. The model of Fig.6 presents all possible sequences of commands describing the system behavior. It is a classical

representation of the dynamic system evolutions. It shows a very large tree or even infinite tree. Thus, exploring all possible program executions is not feasible. We will show in section 6 the weakness of this classical approach. If we consider the representation of the system by a sequence of commands executed iteratively, the previous sequences tree becomes a *repetition of the same subspace pattern* as shown in Fig.7. We will focus in our approach only on this subspace. This represents a specific system command which can be repeated through possible generated sequences.

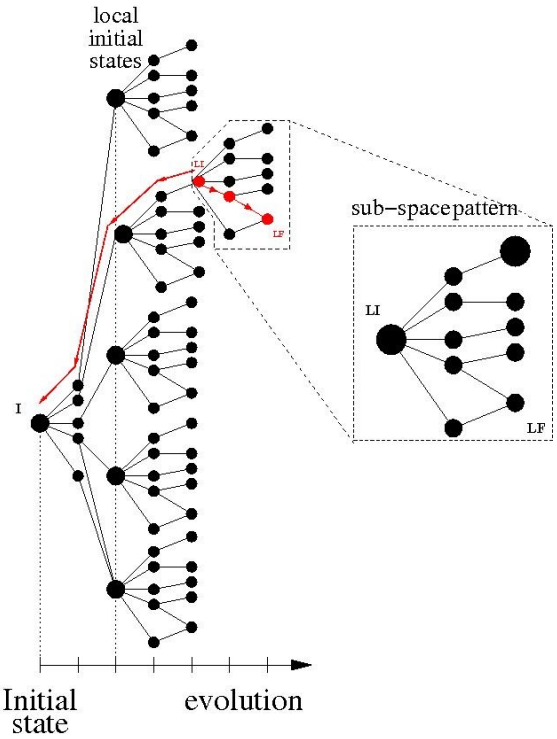

Fig. 7. Model Representation

Each state in the subspace is specified by the symbolic values of the program variables, the path condition and the command parameters (next byte-code to be executed). The path condition represents constraints that should be satisfied by the symbolic values to progress the execution of the current path. It defines the preconditions to successfully follow that path. Our work targets extracting these preconditions from the subspace check. Indeed, we have applied BDD-analysis from the local initial state to local final states of the specified subspace. For each combination of registers, BDD manipulations allow the extraction of the next awaited variables, that lead to the next state and required

preconditions. Outputs are then pushed into a stack, in conjunction with resulting preconditions. Finally, sequence generation pops the constructed stack. Once the necessary preconditions are extracted, a next step is to backtrack the tree until finding the initial sequence fulfilling these preconditions. The backtrack operation is ensured by the compilation process which kept enough knowledge to find later the previous states.

Contrary to the classical sequence generator, our tool constantly generates a tree of pure future states, thus preventing loops from occurring. In other words, previous states always converge to the global initial state. This approach easily favors backtrack execution.

Let's consider the example of Fig.7. Starting from the local initial donated state "LI", we generated all possible paths of tested subspace to reach the final local states using BDDs. Taking into account "LF" (local final state) as a critical final state of the tested system, we executed a backtrack from the "LI" state until covering the sequence that satisfies the extracted preconditions. Assuming state "I" as the final result of this backtrack, the sequence from "I" to "LF" is an example of a good test set. However, considering the representation of Fig.6, a test set from "I" to "LF" will be performed by generating all paths of the tree. Such a test becomes unfeasible if the number of steps to reach "LF" is greatly increased.

4 AUTSEG Description

In this section, we present our testing tool called AUTSEG that implements the approach introduced in the previous section. AUTSEG is particularly used in this paper to test models of various smart cards. Generated automatic test

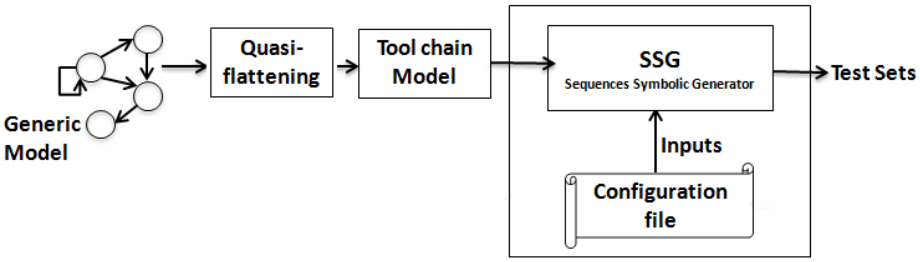

Fig. 8. AUTSEG structure

sets typically must differ according to system input parameters, for example the adopted smart card technology: contact versus contactless. Changing card parameters requires recompiling each new specification separately and re-running the tests. This approach is unrealistic, because this can take many hours or even

days to compile. In addition, this would generate as many models as system types, which can highly limit the legibility and increase the risk of specification bugs. Hence, we have generated a single appropriate global model for all card types and applications, The model's explicit test sets are to be filtered thereafter by AUTSEG. To this end, AUTSEG will query via predefined signals a configuration file specific to each system application. As shown in Fig.8, AUTSEG operation is carried by 5 main entities: (1) The generic model, (2) Quasi-flattening, (3) Compilation, (4) SSG and (5) The configuration file.

After quasi-flattening the hierarchical structure by the "autom-expand" [16] tool implemented according to Algorithm 1, the expanded automaton is compiled separately and linked later with other compiled parallel automata, as shown in Fig.9.

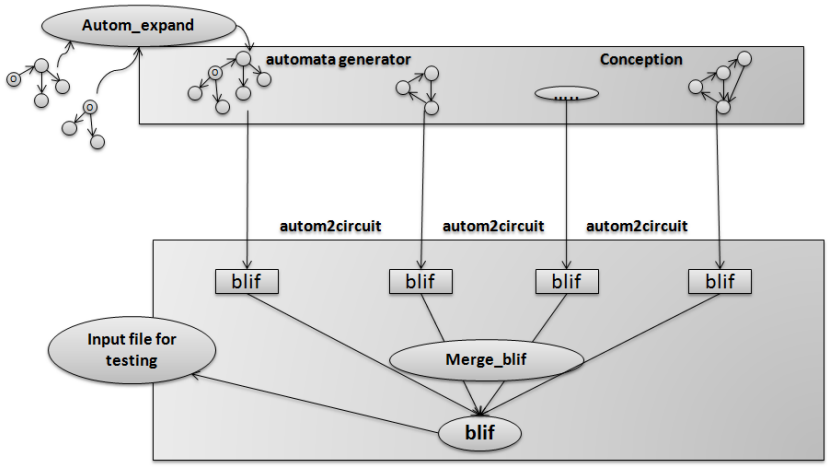

Fig. 9. Model Tool chain

Compilation process is carried out by the "autom2circuit" tool [16], explained in section 3.2. The execution of "autom2circuit" generates a blif file [17] as shown in Fig.9. A blif file is a compact format to express a netlist and is well-suited to represent Boolean equation systems. Using the "merge-blif" tool [16], the generated blif files are concatenated at the end of the compilation process to produce the final global blif file (SSG input file).

In fact, AUTSEG defines two types of preconditions: (1) preconditions related to command parameters as described in section 3.3 and (2) preconditions defined by the configuration file. For a particular test generation, AUTSEG will extract the basic characteristics of the system from the configuration file unit. They are presented as Boolean variables, characterizing the preconditions of the system execution. If preconditions are not satisfied , the tested model will be refined, thus reducing the combinatorial explosion problem during sequence generation.

5 Use case

To illustrate our approach, we studied the case of a contactless smart card designed for the transportation sector. We aim to verify the correct and secure behavior of this card using AUTSEG.

5.1 Smart Card Model

The generic model of the studied smart card is designed from a given transport standard called Calypso. This standard defines 33 commands. The succession of these commands (e.g. Open Session, SV Debit, Get Data, Change Pin) presents possible scenarios of the card operation. All commands have been modeled with the Galaxy tool[16].

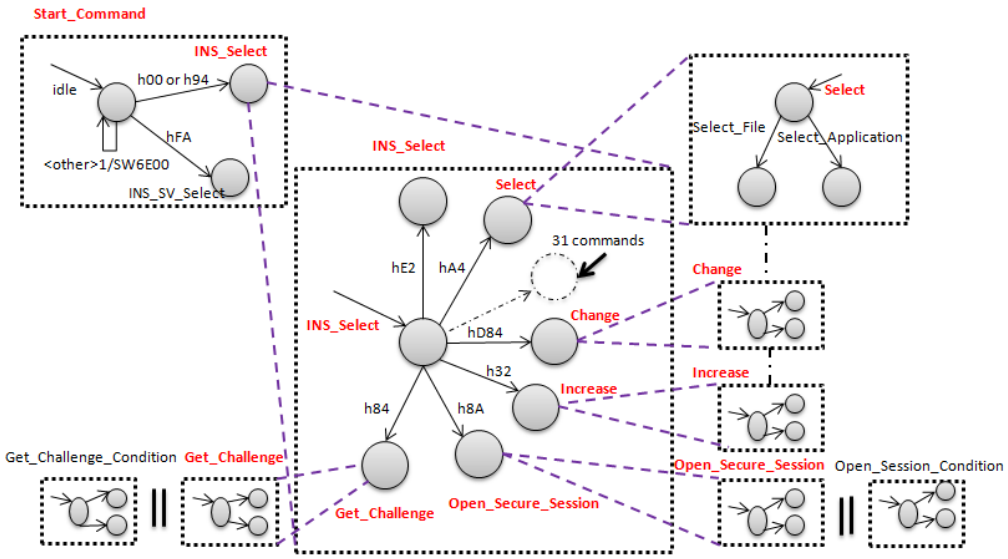

Fig. 10. Calypso smart card model

Galaxy is an automata editor of finite state machines, able to generate four types of automata: simple (basic automata), parallel automata, hierarchical (light Esterel [18]) and syncChart. We chose to use light Esterel (a light version of SyncChart), a synchronous graphical model that integrates high-level concepts of synchronous languages in an expressive graphical formalism. The resulting model presents 52 interconnected automata including 765 states. Forty-three of them form a hierarchical structure. The remaining automata operate in parallel and act as observers for control data of the hierarchical automaton. Fig.10 shows a small part of our model introducing the beginning of card scenarios. Applying

"autom-expand" to the hierarchical automata shows in Fig.11 a flattened structure running in parallel with the observers. Due to "autom-expand", we moved from 477 registers to only 22.

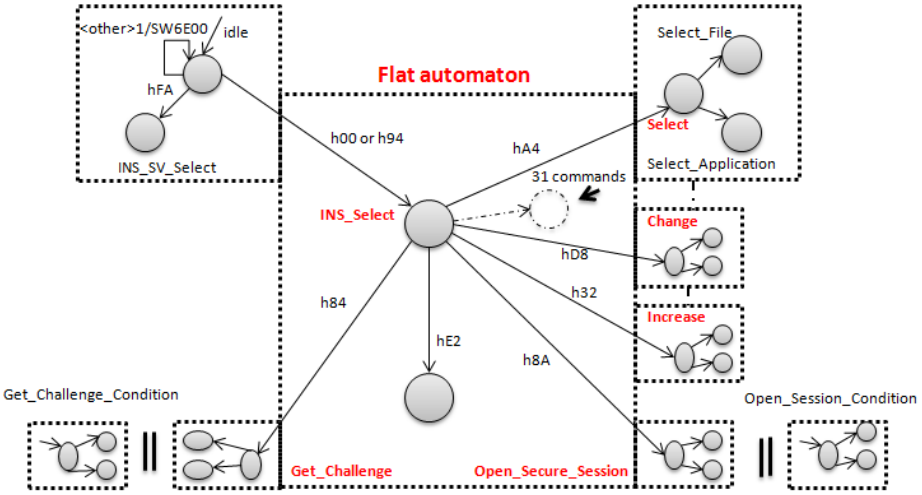

Fig. 11. Calypso Flat Model

5.2 Configuration and Tests

According to the Calypso standard, several types and configurations of the card are defined (contact/contactless, maximum buffer size, etc.). We present these characteristics as preconditions of the system execution ($AUTSEG\_Contact\_Mode$, $AUTSEG\_ch1\_Selected$, etc). The remaining preconditions are established during the execution of Calypso commands. They inform about system status, for instance, $AUTSEG\_V\_Select\_File$ is true if the command "Select File" is executed normally, generating the output code SW9000.

6 Experimental Results

In this section, we show experimental results of applying AUTSEG to the contactless transportation card. We intend to test the security of all possible combinations of 33 commands of the Calypso standard. Each command in the Calypso standard is encoded on a minimum of 8 bytes. We conducted our experiments on a PC with Intel Dual Core Processor, 2 GHz and 8 GB RAM. A classical test of this card can be achieved by browsing all possible paths of the model as represented in Fig.6.

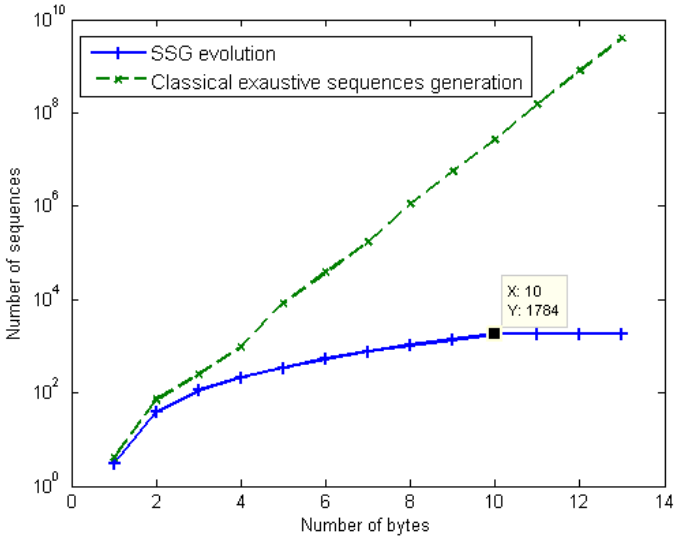

Fig. 12. Combinatorial explosion of classical tests

Such a test shows in Fig.12 an exponential evolution of the number of sequences versus the number of tested bytes. We are not even able to test more than 2 commands of the model. Our model explodes by 13 bytes generating 3,993,854,132 possible sequences.

A second test applies AUTSEG on the card model represented in the same manner as Fig.7. Results show that our approach enables coverage of the global model in a substantially short time. It allows separately testing 33 commands (all the system commands) in only 10 steps, generating a total of 1784 paths. These results are highlighted in Fig.12 by a comparison of our approach to the classical generation method. We note from the AUTSEG curve a lower evolution that stabilizes at 10 steps and 1784 paths, allowing for coverage of all states of the tested model. Covering all states in only 10 steps, our results demonstrate that we test seperately one command (8 bytes) at once in our approach. Few additional bytes (2 bytes in our case) are required to test system preconditions.

We show below an excerpt of generated sequences presenting the two last paths. We observe the extraction of necessary preconditions that should be satisfied for each sequence. $AUTSEG\_Contact\_Mode$ and $AUTSEG\_ch1\_Selected$ are preconditions from the configuration file. They serve to specify the execution context and thus possible resulting sequences. The remaining preconditions ($AUTSEG\_V\_Select\_File$, $AUTSEG\_Verif\_Always\_DF$, etc) will be used to play iteratively the backtrack (e.g, check Setlect File command) until the source sequence is found. We notice the END of sequence generations by 1784 paths, thus covering the entire system execution.

```
AUTSEG TEST SET
-----------------------------------------------
PATH: 1

   ⋮

-----------------------------------------------
PATH: 1783
PRECONDITIONS:
AUTSEG_Contact_Mode
AUTSEG_CH1_Selected
AUTSEG_Verif_Always_DF and AUTSEG_V_Select_File
not AUTSEG_Too_many_modifications_In_session
SEQUENCE:
- h00
- h04
- h00 ---> SW9000 Next_command
-----------------------------------------------
PATH: 1784
PRECONDITIONS:
AUTSEG_CH1_Selected
AUTSEG_Verif_Always_DF and AUTSEG_V_Select_File
AUTSEG_Too_many_modifications_In_session
SEQUENCE:
- h00
- h04
- h00 ---> SW6400 Next_command
-----------------------------------------------

EXPLORATION END FINDING 1784 PATHS...
```

7 Conclusion

We have proposed AUTSEG, an Automatic Test Set Generator for embedded reactive systems. We particularly focused in this paper on systems executing iterative commands. Our tool is able to handle large models, where the risk of combinatorial explosion of states space is important. This has been achieved by essentially (1) providing an algorithm to quasi-flatten hierarchical FSM and reduce the states space, and (2) focusing on pertinent subspaces and restricting the tests. This enables coverage of the global system behavior and generates the list of all possible system evolutions according to the configuration file. Since our tool ensures communication with an interactive specification block, this approach can be adapted to process in parallel several types of system specifications.

In the near future, we will integrate data evaluation during the test process. We aim to use the Linear Decision Diagram LDD [19] to accomplish this. LDD allows the characterization and expression of the given preconditions by numerical constraints. It checks several constraints and concludes about the feasibility of the corresponding sequence. For example, if the union of constraints is unsuccessful by LDD, then we can confirm that tested sequence is impossible and remove it. With this approach, we can reduce the states space and avoid large calculations.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE transaction on Computers **C-35**(8) (1986) 677–691
2. Seljimi, B., Parissis, I.: Automatic generation of test data generators for synchronous programs: Lutess v2. In: Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting. DOSTA '07, New York, NY, USA, ACM (2007) 8–12
3. DuBousquet, L., Zuanon, N.: An overview of lutess: A specification-based tool for testing synchronous software. In: ASE. (1999) 208–215
4. Blanc, B., Junke, C., Marre, B., Le Gall, P., Andrieu, O.: Handling state-machines specifications with gatel. Electron. Notes Theor. Comput. Sci. **264**(3) (2010) 3–17
5. Calam, J.R.: Specification-Based Test Generation With TGV. CWI Technical Report SEN-R 0508, CWI (2005)
6. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: Stg: A symbolic test generation tool. In: TACAS. (2002) 470–475
7. Bentakouk, L., Poizat, P., Zaïdi, F.: A formal framework for service orchestration testing based on symbolic transition systems. Testing of Software and Communication Systems (2009)
8. Xu, D.: A tool for automated test code generation from high-level petri nets. In: Proceedings of the 32nd international conference on Applications and theory of Petri Nets. PETRI NETS'11, Berlin, Heidelberg, Springer-Verlag (2011) 308–317
9. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. ASE '08, Washington, DC, USA, IEEE Computer Society (2008) 443–446
10. Li, G., Ghosh, I., Rajan, S.P.: Klover: a symbolic execution and automatic test generation tool for c++ programs. In: Proceedings of the 23rd international conference on Computer aided verification. CAV'11, Berlin, Heidelberg, Springer-Verlag (2011) 609–615
11. André, C.: Representation and analysis of reactive behaviors: A synchronous approach. In: Computational Engineering in Systems Applications (CESA), Lille (F), IEEE-SMC (July 1996) 19–29
12. Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program. **19**(2) (November 1992) 87–152
13. Paiva, A.C.R., Tillmann, N., Faria, J.C.P., Vidal, R.F.A.M.: Modeling and testing hierarchical guis. In: Proc.ASM05. Universite de Paris 12. (2005) 8–11
14. Wasowski, A.: Flattening statecharts without explosions. SIGPLAN Not. **39**(7) (Jun 2004) 257–266
15. Chiuchisan I., Potorac A.D., G.A.: Finite state machine design and vhdl coding techniques. In: 10th International Conference on development and application systems, Suceava, Romania, Faculty of Electrical Engineering and Computer Science (2010) 273–278
16. Gaffé, D.: Research web site. http://sites.unice.fr/dgaffe/recherche/research.html
17. Berkeley University: Berkeley logic interchange format (blif). (1998)
18. Ressouche, A., Gaffé, D., Roy, V.: Modular compilation of a synchronous language. In Lee, R., ed.: Soft. Eng. Research, Management and Applications, best 17 paper selection of the SERA'08 conference. Volume 150., Prague, Springer-Verlag (August 2008) 157–171
19. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear arithmetic. In: FMCAD, IEEE (2009) 53–60